ECE 385

Fall 2024
Experiment 2

Lab 2 Report

Yousef Khojah (yousefk2)
Mohammed Alnasser (ma138)

Introduction

In this lab, we are designing and building a bit-serial logic operation processor with the use of several logic gates, multiplexers, and a counter. The lab consists of two parts. In the first, we designed and built a 4-bit physical circuit. In the second, we extended the design on Vivado using SystemVerilog to an 8-bit processor. The processor has four main units: register, computation, routing, and control. The register unit stores the loaded or computed result of inputs A and B. The computation unit can compute the result of the eight logic operations AND, OR, XOR, 1, NAND, NOR, XNOR, and 0. The result then goes to the routing unit, where you can control if the result should be stored in either A or B, keep A and B the same, or swap A with B. This whole process is controlled through the Finite State Machine (FSM) of the control unit.

Operation of the Logic Processor

To load a value in either A or B, you should first flip the 4 switches (D_3 - D_0) on the FPGA for the wanted value, then flip the switch LOAD A/B as shown in Fig.1. Then, to perform an operation, you want to set up the desired computation and route you wish for the result. The 3 switches $F_2F_1F_0$ decide which operation will be performed, and the switches R_1R_0 control the route. After setting the operation and route, you can hit EXECUTE, and the new result can be seen on the registers A and B.

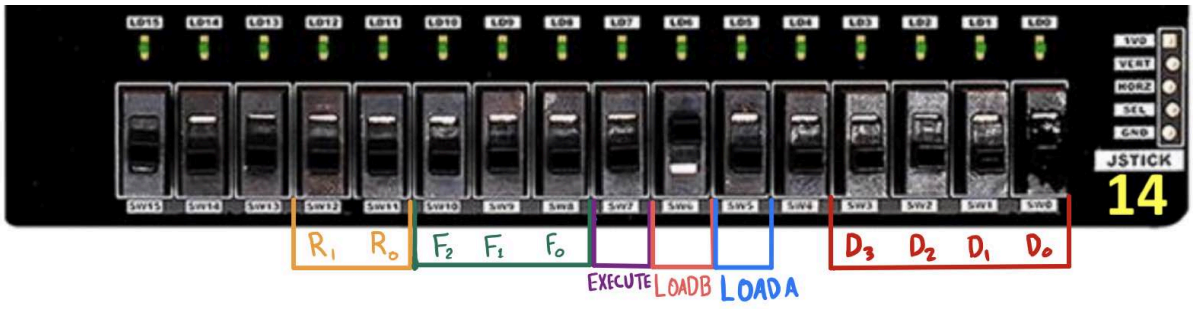

Fig.1: View of the circuit's input switches

Function Selection Inputs			Computation Unit Output
F2	F1	F0	f(A, B)
0	0	0	A AND B
0	0	1	A OR B
0	1	0	A XOR B
0	1	1	1111
1	0	0	A NAND B
1	0	1	A NOR B
1	1	0	A XNOR B
1	1	1	0000

Routing Selection		Router Output	
R1	R0	A*	B*
0	0	A	B
0	1	A	F
1	0	F	B
1	1	B	A

Table.1: Table for the F and R signals that control the computation and routing units

Description:

- Register Unit:

  The register unit consists of two 4-bit shift registers, RegA and RegB, which hold the values for A and B. The values of the registers are always displayed on the multisegment LEDs. The inputs of the registers are either loaded by the user or serially from the routing unit. The control unit's output S combined with Load A and Load B signals go through the combinational logic block, as shown in Fig.2, to control whether the registers should hold their value, shift, or load D3-D0 from switches. We can control this through S1 and S0; whenever S1S0 = 00, that's the hold state. S1S0 = 01, that's the shift state. S1S0 = 11, that is the load state. The expressions for both S1 and S0, alongside the truth table, can be found in Table.2 below.

LD A/B	S	S1	S0
0	0	0	0
0	1	0	1
1	0	1	1
1	1	0	1

Table.2: Truth table for S1 and S0 inputs of the register

S1 = LDA\B*S' S0 = LDA/B+S

- Computation Unit:

  The computation unit takes the 1-bit input from RegA and RegB and the select inputs $F_2F_1F_0$. As can be seen in Table. 1, the unit will output the function f(A, B), which is chosen in a multiplexer by the select inputs, and it will output inputs A and B unchanged.

- Routing Unit:

  The outputs f(A, B), A, and B from the computation unit will go through a multiplexer in the routing unit to choose the route in which it will be stored. This decision is based on the select inputs R_1R_0. As can be seen in Table.1, the result can be stored in either RegA or RegB while keeping the other input the same, or RegA and RegB keep their values in the original place or swapped.

- Control Unit:

  The control unit has inputs LOAD A, LOAD B, EXECUTE, and a clock signal as shown in Fig.2. The LOAD A and LOAD B are responsible for parallel loading the RegA and RegB, respectively. Similarly, the EXECUTE input is responsible for executing the selected operation on the stored inputs in RegA and RegB and sending the selected path to the routing unit to store the result. The clock is used for the registers and the FSM's flip-flop so that the computation cycle works in an instant manner, but enabling the user to debug and observe each step.

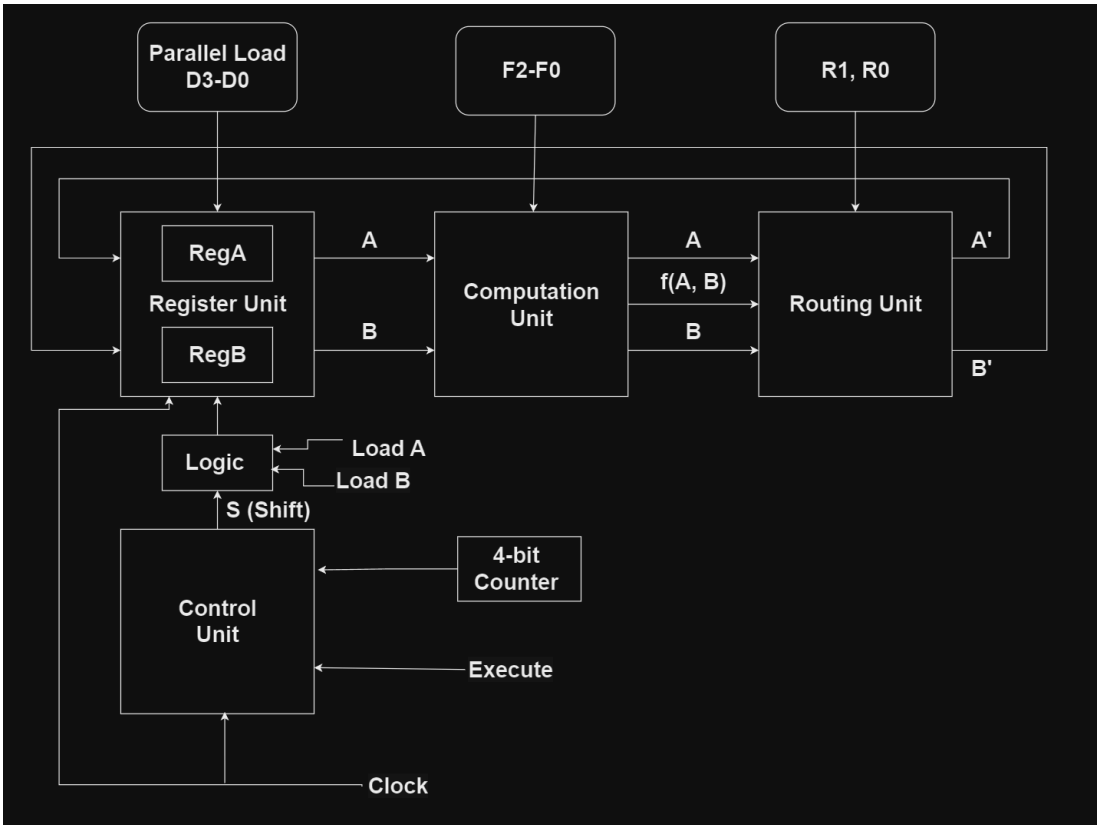

Fig.2: High-level block diagram for the 4-bit processor

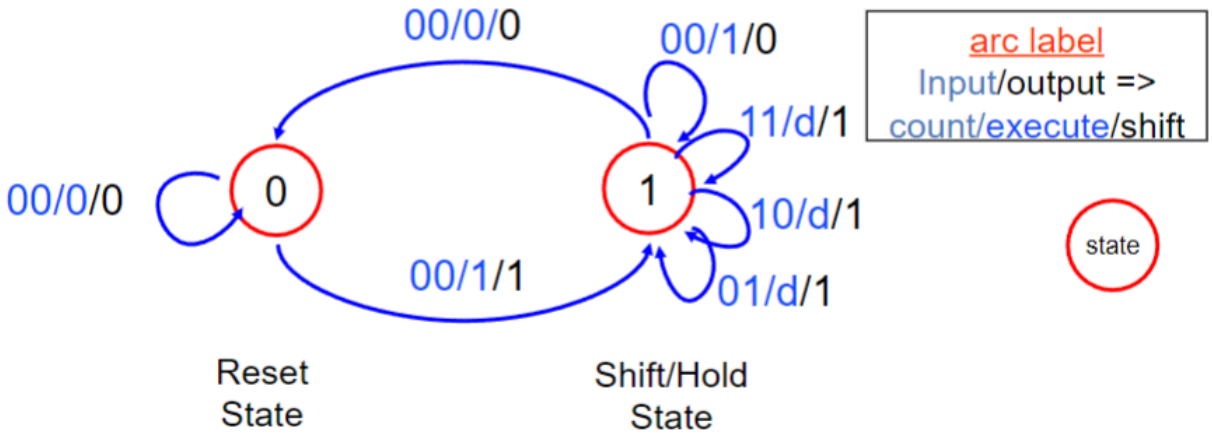

Fig.3: State transition diagram for the control unit FSM

➢ FSM:

The most important part of the control unit is the FSM, which controls the states of the whole processor. In our design, we used a Mealy machine, which has fewer states than its Moore counterpart. We had only two states, which can be described as the reset state and the shift/hold state, as shown in Figure 4. There are four signals that control the state transition, which are the input EXECUTE, a single-bit state representation Q, and the least two significant bits of a counter C1C0, as can be seen from Table.3 below. The output of this FSM is the signal S, which stands for shift and the new state representation Q^+, the expressions for both outputs can be found in Fig.3. The output S is what controls the registers RegA and RegB. Whenever S is 1, the registers will start to shift. Otherwise, it will depend on the signals LOAD A and LOAD B to load or just hold the value.

Exec. Switch ('E')	Q	C1	C0	Reg. Shift ('S')	Q^+	$C1^+$	$C0^+$
0	0	0	0	0	0	0	0
0	0	0	1	d	d	d	D
0	0	1	0	d	d	d	D
0	0	1	1	d	d	d	D
0	1	0	0	0	0	0	0
0	1	0	1	1	1	1	0
0	1	1	0	1	1	1	1
0	1	1	1	1	1	0	0
1	0	0	0	1	1	0	1
1	0	0	1	d	d	d	D
1	0	1	0	d	d	d	D
1	0	1	1	d	d	d	D
1	1	0	0	0	1	0	0
1	1	0	1	1	1	1	0
1	1	1	0	1	1	1	1
1	1	1	1	1	1	0	0

Table.3: Control unit state transition table using the Mealy state machine

Circuit Schematic

Next State Representation Q+				
EQ\C$_1$C$_0$	00	01	11	10
00	0	x	x	x
01	0	1	1	1
11	1	1	1	1
10	1	x	x	x
$Q^+ = E + C_0 + C_1$				

Shift Signal (S)				
EQ\C$_1$C$_0$	00	01	11	10
00	0	x	x	x
01	0	1	1	1
11	0	1	1	1
10	1	x	x	x
S = E$\overline{Q}$ + C$_0$ + C$_1$				

Fig.4: K-map and expressions for both S and Q^+

The K-maps in Fig.4, in addition to the truth tables in Table.2 and Table.3 were the ones that were used in designing the lab and creating the schematic in Fig.5. However, the expressions drawn from them cannot be used directly in the circuit as we don't have any AND or OR gates. NAND and NOR gates were used instead, even when implementing the functions AND and OR in the computation unit. The modifications that were made to the expressions are as follows:

Q^+= E + C$_0$ + C$_1$ = $\overline{\overline{(E + C_0 + C_1)}}$ ⇒ A 3-input NOR gate with an inverter was used

S = E$\overline{Q}$ + C$_0$ + C$_1$ = $\overline{\overline{(E\overline{Q} + C_0 + C_1)}}$ ⇒ A 3-input NOR gate with an inverter was used

S$_1$ = LOAD A\B * $\overline{S}$ = $\overline{\overline{(LOAD\ A\backslash B\ *\ \overline{S})}}$ ⇒ A 2-input NAND gate was used twice

S$_0$ = LOAD A\B + S = $\overline{\overline{(LOAD\ A\backslash B\ +\ \overline{S})}}$ ⇒ A 2-input NOR gate was used twice

When designing the circuit, we had several considerations regarding switches and chips. There were two types of register chips, and both would probably work, but one was suited better for the lab's purpose as it had only left shift instead of the other one, which had right and left with input to choose which one. Also, we were considering the kit's switches or the FPGA one and ended up using the FPGA's because the kit's gave unstable outputs. Lastly, in the computation unit, we chose to work only with four operations, then XOR the result with F$_2$ so you can have the function or its complement. This approach uses less space but could be trickier to implement.

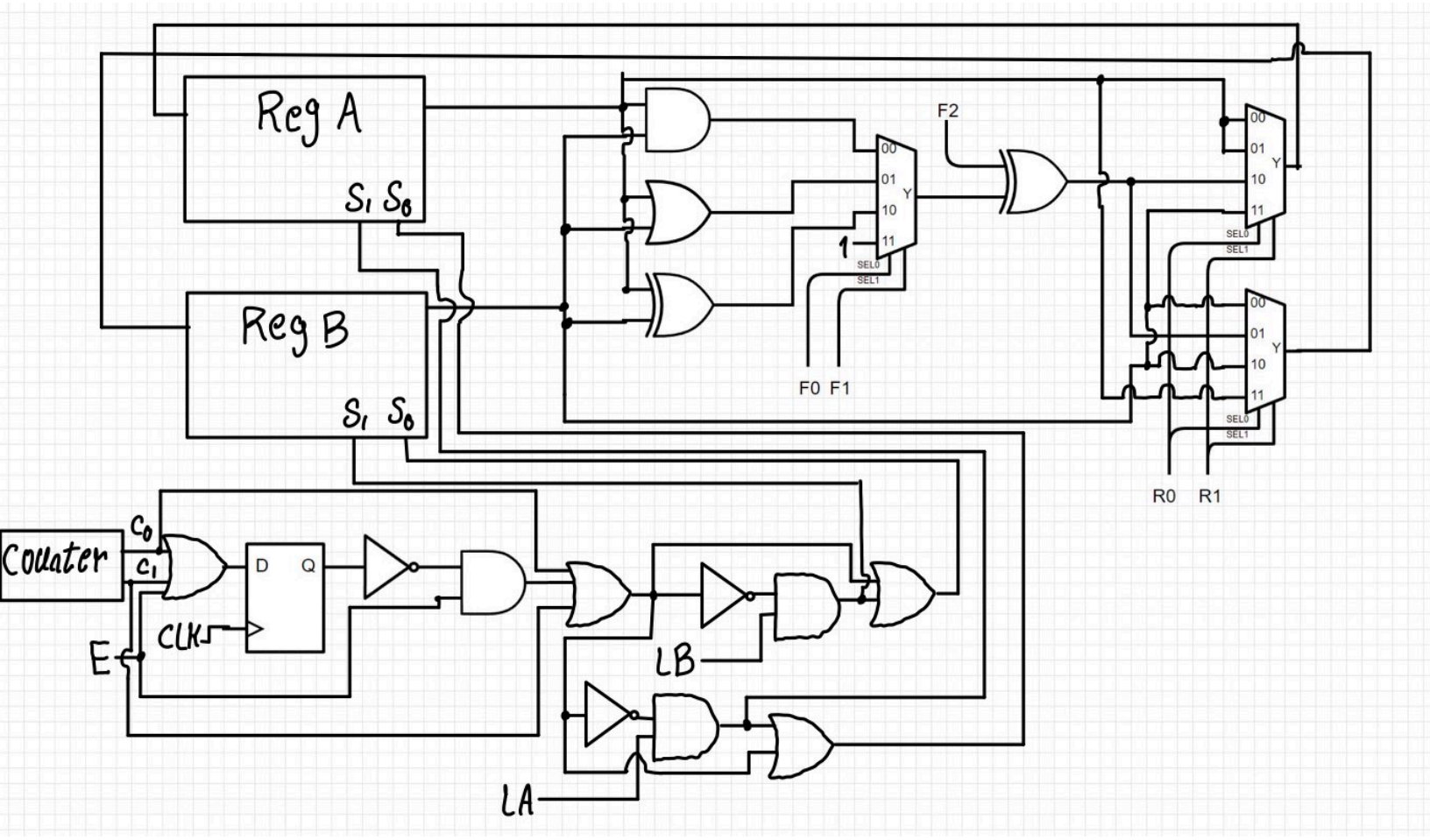

Fig.5: Circuit schematic for the 4-bit processor

Breadboard Layout Sheet

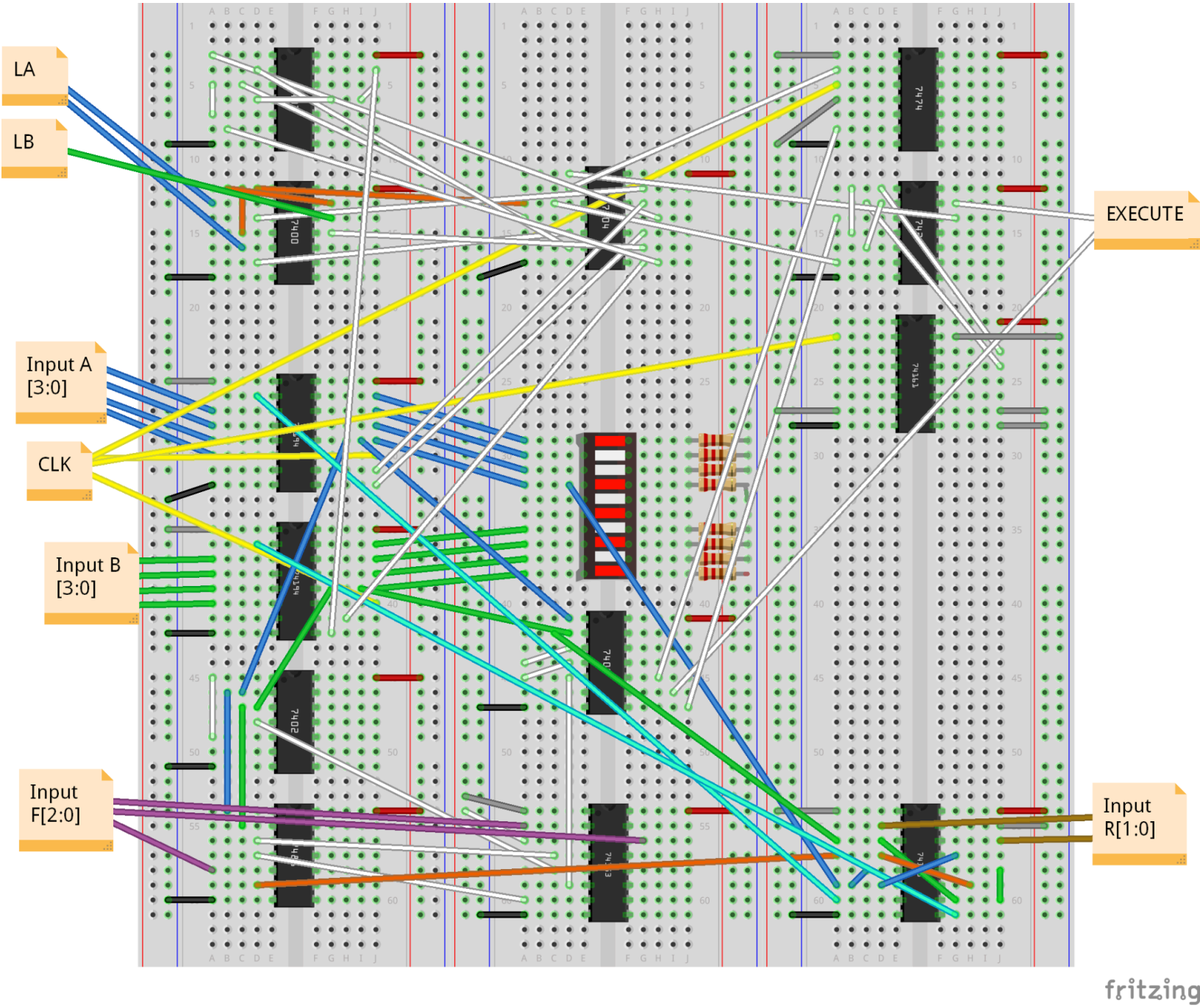

Fig.6: Breadboard layout for the 4-bit processor using fritzing

8-bit logic processor on FPGA

- ### Processor.SV:

  Inputs: Clk, Reset, Load A, Load B, Execute, [7:0] Din, [2:0] F, [1:0] R
  Outputs: [3:0] LED, [3:0] hex_grid, [7:0] Aval, [7:0] Bval, [7:0] hex_seg
  Description: This is the top-level module where all instantiations happen.
  Purpose: This file is the top-level design source, as it instantiates all of the other modules and initializes all inputs and outputs. A few necessary changes were made to this file. The initialization of A, B, Din, Aval, and Bval was extended from 4 bits [3:0] to 8 bits[7:0]. Also, LEDs on the FPGA board were modified to accommodate more hex drivers so that each LED represents a hexadecimal in A or B. Lastly, the Din_sync debounce button was extended to 8 bits.

- ### Register_unit.SV:

  Inputs: Clk, Reset, A_In, B_In, Ld_A, Ld_B, Shift_En, [7:0] D
  Outputs: A_out, B_out, [7:0] A, [7:0] B
  Description: This is a positive-edge triggered 8-bit register with asynchronous reset and synchronous load. When Load is high, data is loaded from Din into the register on the positive edge of Clk. When the Shift is high, the contents of the register right shift on the rising edge of the Clk.
  Purpose: This module represents the registers RegA and RegB. The inputs A, B, and the parallel load D were all extended to 8 bits instead of 4. Moreover, there was a part of the code that specifies which bits get shifted [3:0] Data_Out_d, Data_Out [3:1]; these were extended to 8 bits as well, [7:0] and [7:1], respectively.

- ### Compute.SV:

  Input: A_In, B_In, [2:0] F
  Output: A_Out, B_Out, F_A_B
  Description: This is an 8:1 multiplexer with F[2:0] as its select inputs. Its inputs are the following logic operations between A and B, starting from F = 000, AND, OR, XOR, 1, NAND, NOR, XNOR, 0.
  Purpose: This module computes the desired logic operations in the design of our processor. Since this file deals with the inputs in a bit-by-bit manner, we didn't have to change anything in it.

- ### Router.SV:

  Input: [1:0] R, A_In, B_In, F_A_B
  Output: A_Out, B_Out
  Description: This consists of two 4:1 multiplexers with select inputs R[1:0]. One multiplexer will be used for each register to choose what will be stored next.

Purpose: This file chooses the route the inputs and the operation result take. This file also deals with its inputs bit-by-bit, so nothing was modified in it.

- ## Control.SV:

Input: Clk, Reset, LoadA, LoadB, Execute
Output: Shift_En, Ld_A, Ld_B
Description: This is a positive-edge-triggered flip-flip with combinational logic to calculate the shift signal S and the input of the flip-flop.
Purpose: The control unit organizes the operation of our processor, and some modifications are needed. Since the processor was extended to 8 bits, the counter also needed some extension to increase the number of states in the FSM. As a result, the number of bits we became interested in in the counter increased to 3 instead of 2 for 8 states instead of 4.

- ## HexDriver.SV:

Input: clk, reset, [3:0] in[4]
Output: [7:0] hex_seg, [3:0] hex_grid
Description: This module acts as a converter for 4 input nibbles, each into a 7-segment display representation.
Purpose: This file is responsible for formatting and converting an input to be displayed on the LED so that each LED corresponds to one hexadecimal. This file is not affected by the extension to 8 bits, so no changes were made to it.

- ## Synchronizers.SV:

Description: This acts as a flip-flip to hold the button's value.
Purpose: This file had multiple sync_debounce modules for each button used on the FPGA. The purpose of this is to debounce switches since mechanical switches cause glitches during the switch due to contact bounce.

RTL Block Diagram

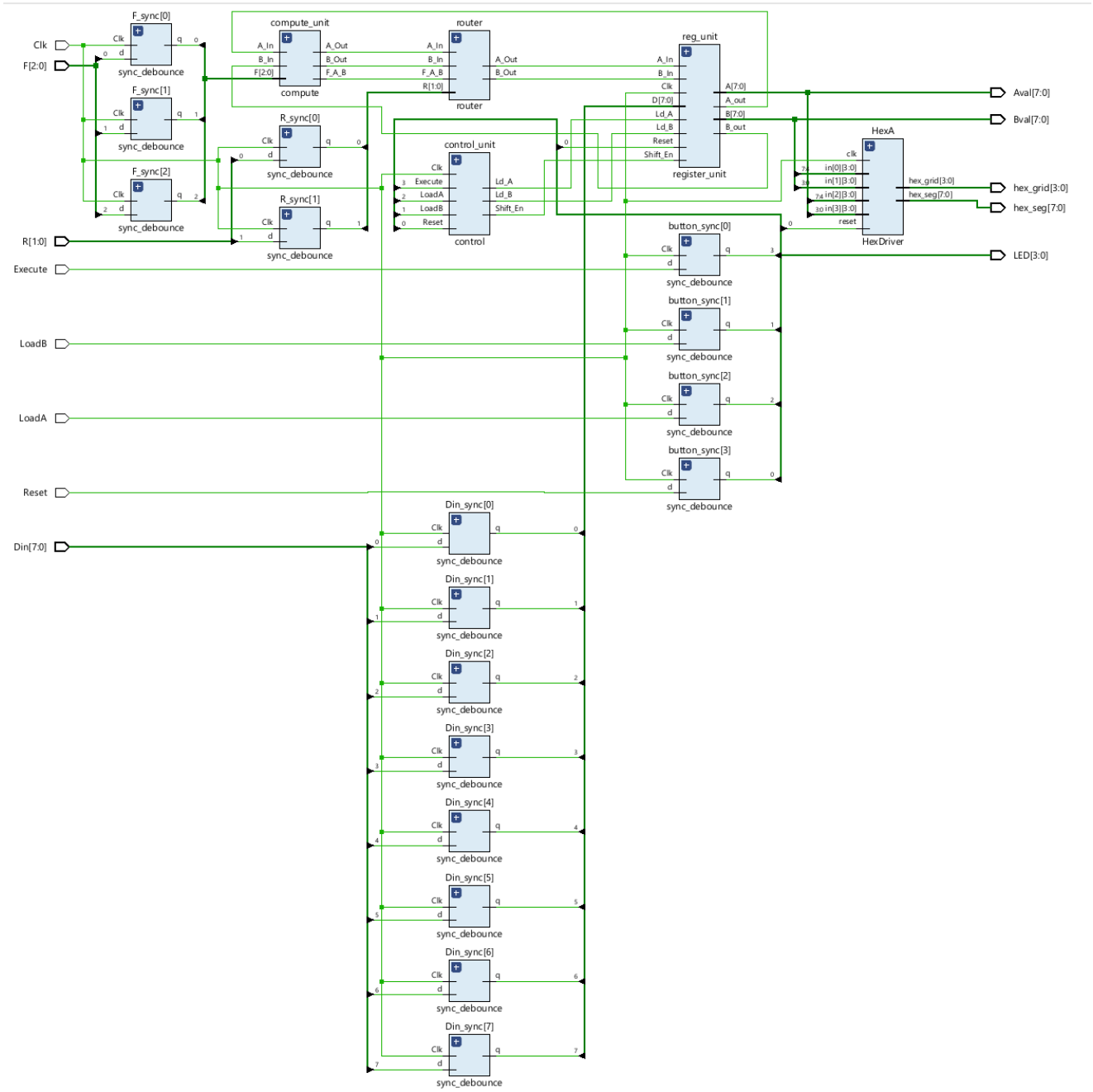

Fig.7: RTL block diagram for the 8-bit processor from vivado

Simulation of The Processor

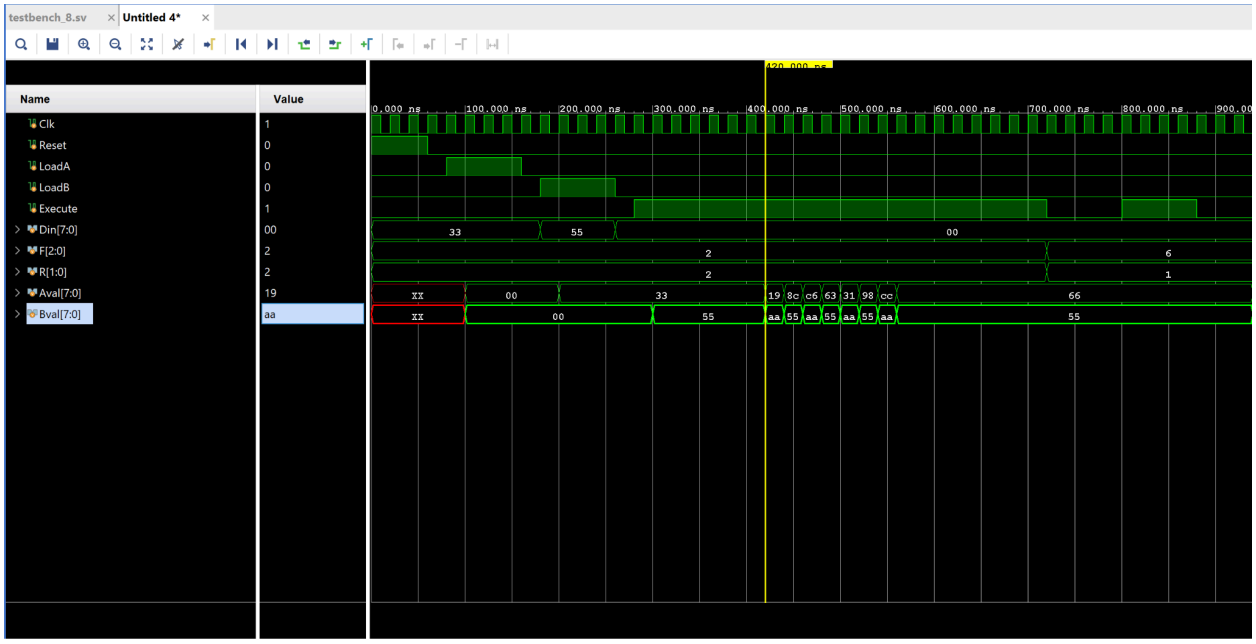

Fig.8: Behavioral Simulation of the 8-bit Processor

Fig.8 shows the behavioral simulation of the 8-bit processor with various signals. As seen in the waveform, when LoadA was pressed, Aval got loaded with Din (x33). Similarly, when LoadB was pressed, Bval got loaded with the updated value of Din (x55). We can also see that the F[2:0] signal is set to 2, meaning that the operation performed is XOR, as shown in Table.1. Moreover, the R[1:0] signal is set to 2, which means that the result of XORing A and B will be stored in register B; meanwhile A will remain the same. Additionally, we can observe the sequence of right shifts happening at 420ns, which is required to give us the correct result. In summary, this behavioral simulation shows the result of A XOR B and stores it in register B while preserving register A.

Vivado Debug Core

Vivado debug core, as the name implies, is a tool used to help debug the implemented design on the FPGA hardware. The main difference between it and behavioral simulation is that Vivado debug core allows one to directly debug the implemented design using FPGA hardware, meaning that the debug core allows for real-time debugging. The steps followed to set up a debug core on Vivado are as follows:

Step 1) Go to synthesis -> open synthesized design -> set up a debug core

Step 2) Click next until you see **Nets to Debug** page. In this page, you'll be able to choose what signals to monitor by clicking on the "Find Nets to Add" button on the bottom left. For example, you can choose signals A and B to see how they change once Execute is pressed.

Step 3) Set sample size

Step 4) Click Finish

Step 5) Make sure inputs/outputs are assigned

Step 6) Trigger Setup. It is essential to set up a trigger to tell the debug core when to start monitoring. In our case, we set the trigger to occur when the most significant bit of LED_OBUF[3:0] equals one, which is when execute is pressed.

Step 7) Click run and use inputs on the FPGA board.

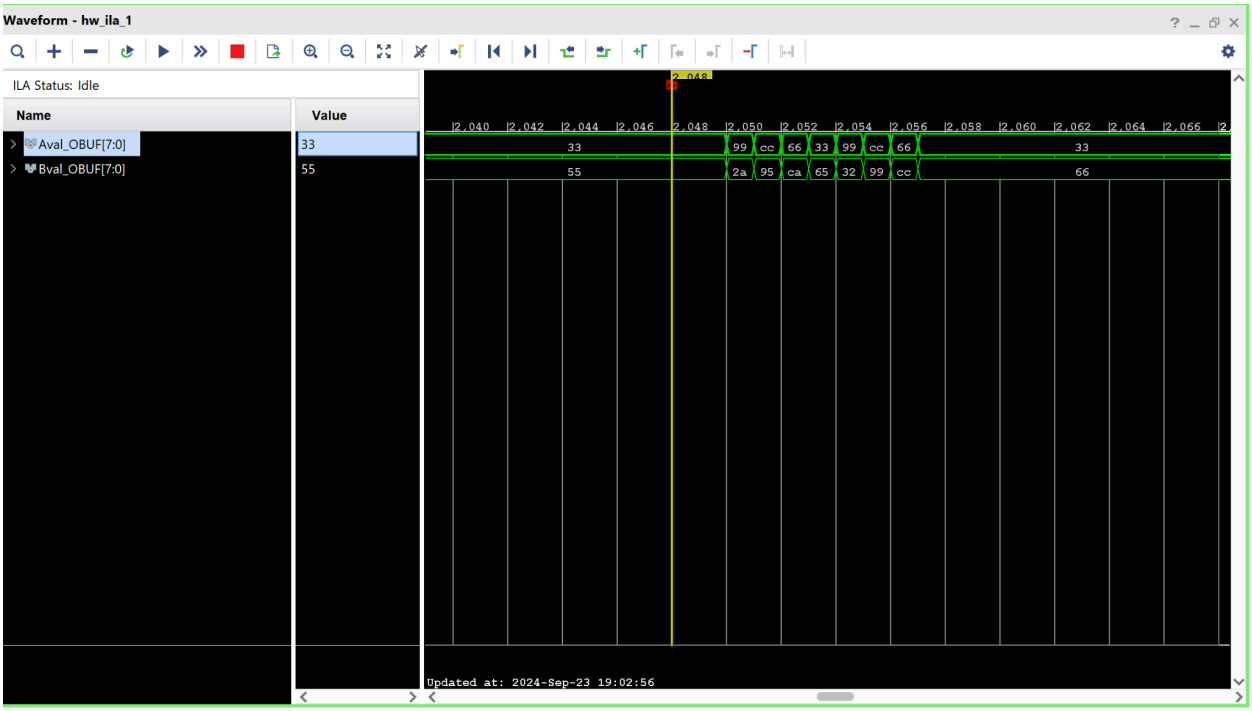

Fig.9: Output of the debug core trace performing an XOR operation and save the result in register B

Bugs Encountered

We have encountered three main problems in this lab. In part one, we used the multisegment switches that came with the kit. However, these switches gave unstable signals when displayed on the LED, so instead, we tried the FPGA board's switches, and they worked out. We had to wire every input to the board pins, as seen in Fig.1 . Secondly, in part one as well, our circuit had problems with floating inputs, and this is because the chips used had some essential inputs that we were not aware of and were not described clearly in the datasheet, such as the strobe G input in multiplexers which needed to be grounded. Lastly, in part two, we

extended all that appeared to be necessary to change in all files, but for some reason, the two most significant hexadecimals were not working correctly. There seemed to be a line of code that was responsible for what bits get shifted, and it was on [3:1], so we just needed to change that to [7:1].

Conclusion

In this lab, we successfully designed and implemented a bit-serial logic operation processor through both physical circuit construction and an FPGA-based extension composed of 4 main units: a register unit, a computation unit, a routing unit, and a control unit. The processor was capable of parallel loading RegA and RegB with input D[3:0], performing eight basic logic operations AND, OR, XOR, 1, NAND, NOR, XNOR, and 0, and routing the computed results between two registers determined by the inputs F[2:0] and R[1:0]. The computation cycle was organized using a Mealy Finite State Machine (FSM). The two states of the FSM depended on the EXECUTE signal, the state representation Q, and the counter's 2 least significant bits C_1C_0. We built a 4-bit processor physically, using multiplexers, counters, registers, a flip-flop, and logic gates, and then extended this design to an 8-bit processor using SystemVerilog on Vivado, which required some level of understanding of the syntax. By utilizing a modular approach, we were able to ensure flexibility in both the design and debugging phases, allowing for easier troubleshooting.

Post-lab Questions

- **Describe the simplest (two-input one-output) circuit that can optionally invert a signal (i.e., one input determines if the output is equal to the other input or equal to the other input inverted). Explain why this is useful for the construction of this lab.**

  A two-input XOR gate is the simplest logic circuit that can invert a signal conditionally. One input is the data we want to pass, and the other is the signal that controls inverting the input. This is particularly useful for the computation unit part of our lab since half of the operations are basic logic gates, such as AND and OR, and the other half is just the negation of these operations, such as NAND and NOR. So, instead of using an 8:1 Multiplexer to implement the computation unit, we used a 4:1 Multiplexer and XORed the output with the F2 signal. This way, we can get the basic logic operations when F2 is set to 0, and when we want the negation, we just set F2 to 1. This approach reduced the logic chips we needed to use to implement the computation unit, leading to an optimized design.

- **Explain how a modular design such as that presented above improves testability and cuts down development time.**

  The modular design helps keep the relevant parts of the code together so that it is easier to develop a certain area or return to it easily. Another important aspect is that the

modular design makes the debugging process much faster in terms of identifying the problem and then resolving it.

- **Discuss the design process of your state machine, what are the tradeoffs of a Mealy machine vs a Moore machine?**

  The process starts with writing down all the needed states for the design and then trying to optimize it to the maximum capacity. Since a Moore machine depends only on its states, it makes sense to start the design this way. The necessary states are a reset state, a halt state, and four shift states. This Moore machine can be further reduced to three states by combining all the shift states into one state with three transitions on itself. The Moore machine is simpler to implement and design because it is more intuitive. This design will be additionally optimized, but it will not depend only on the states, which means it will become a Mealy machine. The shift and halt states can be combined, but the transitions will now depend on EXECUTE, the state Q, and the counter bits, which is what we implemented in our design. The Mealy machine offers a more efficient design but is more complex as well.

- **What are the differences between vSim and Vivado Debug Cores? Although both systems generate waveforms, what situations might vSim be preferred and where might debug cores be more appropriate?**

  The main difference between it and behavioral simulation is that Vivado debug core allows one to directly debug the implemented design using FPGA hardware, meaning that the debug core allows for real-time debugging.